NAS Technical Report; NAS-06-019
December 2006

Columbia Application Performance Tuning Case Studies

Johnny Chang
NASA Advanced Supercomputing Division
Computer Sciences Corporation
NASA Ames Research Center
Moffett Field, California 94035-1000, USA
jchang@mail.arc.nasa.gov

Abstract: This paper describes four case studies of application performance enhancements on the Columbia supercomputer. The Columbia supercomputer is a cluster of twenty SGI Altix systems, each with 512 Itanium 2 processors and 1 terabyte of global shared-memory, and is located at the NASA Advanced Supercomputing (NAS) facility in Moffett Field. The code optimization techniques described in the case studies include both implicit and explicit process-placement to pin processes on CPUs closest to the processes' memory, removing memory contention in OpenMP applications, eliminating unaligned memory accesses, and system profiling. These techniques enabled approximately 2- to 20-fold improvements in application performance.

Key words: Code tuning, process-placement, OpenMP scaling, memory contention, unaligned memory access.

1 Introduction

An integral component of the support model for a world-class supercomputer is the work done by the applications support team to help the supercomputer users make the most efficient use of their computer time allocations. This applications support involves all aspects of code porting and optimization, code debugging, scaling, etc. Several case studies derived from our work in helping users optimize their codes on the Columbia supercomputer have been presented at both the 2005 [1] and 2006 [2] SGI User Group Technical Conference. This paper describes four of those case studies.

First, we present a brief description and history of the Columbia supercomputer, which also sets the terminology used throughout the paper. For example, the definition of a "node" can be different for different people. The first two case studies deal with process-placement – the first one does the pinning implicitly via the "dplace" command, and the second does it explicitly by calling the "cpuset_pin" function from within the user code. The third case study deals with OpenMP scaling on the SGI Altix, and the fourth on eliminating unaligned memory accesses from user codes.

2 Columbia supercomputer

The Columbia supercomputer is a cluster of twenty SGI Altix systems, each with 512 Intel Itanium 2 processors and 1 terabyte of global shared-memory. Twelve of these systems are of the SGI Altix 3700 series [3] and the other eight are of the newer SGI Altix 3700 BX2 systems. Four of the BX2's are interconnected via NUMAlink 4 into a 2048-processor capability system. In the summer of 2004, as each additional 512-processor system was delivered, it was assembled in one-day, a set of diagnostics was run on the second day, and on the third day, the machine was available for user applications. By October, 2004, NASA had enough systems to obtain a LINPACK number [4] that placed Columbia number one in the world. That announcement [4], however, was short-lived as nine days later, IBM announced [5] a LINPACK number that exceeded even the theoretical peak of Columbia. In the past 3 semi-annual rankings on the Top500 list [6], Columbia's 51.87 TFlops LINPACK number places it 2nd, 3rd, and 4th on the Nov. 2004, June 2005, and Nov. 2005 rankings, respectively.

The basic computational building block of the SGI Altix 3700 system is the C-brick, which consists of two nodes connected to each other via a NUMAlink 4 interconnect. Each node contains two processors, which share a front-side bus connection to a single on-node memory via an ASIC called the Super Hub or

SHUB for short. The SHUB is also used to connect processors on a node to processors outside the C-brick via router- or R-bricks or directly through other SHUBs. For the Altix 3700 series, the network connecting outside the C-brick is NUMAlink 3. The BX2 systems differ from the earlier 3700 series in that all the nodes are interconnected via the NUMAlink 4 interconnect, which has twice the bandwidth of NUMAlink 3. Although each processor on an SGI Altix has access to memory on all other nodes, there is a performance penalty associated with accessing remote memory especially as the number of router hops increases. Removing or reducing this remote memory access by increasing local memory access is a common theme in three of the case studies on performance enhancement.

3 Case Study 1: Implicit process-placement

In the first case study, one of our researchers wanted to create an aeroelastic stability derivative database by running multiple copies of the Overflow code in serial mode. When he ran one copy of the executable, it took 12 minutes to run. With 128 copies, it took 30 minutes, and with 500 copies, it took more than 6 hours to complete. What's going on here? Ideally, one would like all 500 copies running on 500 CPUs to finish at the same time -- in 12 minutes.

What was happening here is that the kernel started several processes on the same set of CPUs causing massive contention. Eventually, when processes were moved to idle CPUs, their memory was not moved, so those migrated processes would access non-local memory. To avoid these problems, we have to start each process on a separate CPU and pin them there to avoid their migrating to other CPUs. This is accomplished with the dplace command. In the script below, we show the modified parts of the script in red. First, we set n – the relative CPU number – to zero. Then we loop over i, j, and k for the 10 by 10 by 5 or 500 cases. For each case, we "cd" to a subdirectory, remove the previous output file, and run the Overflow program with the dplace command, putting it in the background. "n" is then incremented for the next point in the database and so forth. There is a "wait" at the end to wait for all the backgrounded processes to complete before proceeding.

```
# set the relative cpu number (first one at 0)
set n = 0

foreach i ( 1 2 3 4 5 )
foreach j ( 0 1 2 3 4 5 6 7 8 9 )
foreach k ( 0 1 2 3 4 5 6 7 8 9 )

cd CASE$i$j$k
/bin/rm -f  boost$i$j$k.out
cp rgrid$i.dat grid.in
cp case$j$k over.namelist

dplace -c $n ./overflow  > boost$i$j$k.out &

# increment relative cpu number
@ n++
cd ..
end
end
end
wait
```

With process-pinning, the 500 process job took 17 minutes to complete instead of over 6 hours. That's a 21x speed-up. Well, one might wonder why it took 17 minutes instead of 12. There are two reasons for this. First, there is some memory contention because the two processes on a node share a single front-side bus to the memory. Second, with all 500 processes reading and writing to the same filesystem, there is also disk contention as well. In any case, the user was very happy to get the 21x speed-up.

4 Case Study 2: Explicit process-placement

For the second case study, we look at one of the optimization steps for the NASA finite-volume General Circulation Model (fvGCM) code. This is a global weather modeling code where researchers have been cranking up the resolution over the past few years [7]. At the $1/8^{th}$ degree resolution, that translates to about a 10 km grid spacing along the equator. With so many grid points, the memory requirements of the code were huge. The code is written in a hybrid MPI+OpenMP programming paradigm.

When we first started running the code, the kernel would kill the job because it tried to access more memory than what's available in the cpuset. A cpuset is a set of CPUs that's allocated to the job by the batch queueing system. By trial-and-error, we figured out that a 20 MPI by 4 OpenMP case (an 80 processor job) needed the memory of 268 CPUs to run. We used SGI's message passing library, which is already tuned for the Altix machine. There is an environment variable MPI_OPENMP_INTEROP that one can set to improve the placement of processes and threads. In particular, if one has MPI_OPENMP_INTEROP set, and OMP_NUM_THREADS is set to 4, then when MPI starts up, the MPI processes are placed 4 CPUs apart to leave space for the OpenMP threads spawned by each process. This is the right thing to do because then, the threads would be accessing the local memory that's closest to the process that spawned them. Unfortunately, for this case, there were still lots of non-local memory accesses. If you divide the memory of 268 CPUs by 80 processes and threads, you see that each thread needs the memory of approximately 3 CPUs. So, to improve the memory locality, you'll want to space the threads 3 CPUs apart. This is done by explicitly calling the process-pinning function from within the fvGCM code. The code modifications are relatively straightforward: right after the MPI_Init, MPI_Comm_rank and MPI_Comm_size function calls, one puts in an OpenMP parallel region, which does nothing but determine which thread it is in the series and pins the thread to the appropriate relative CPU.

Here's the interface to the process-pinning function under the Linux 2.6 kernel:

```
#include <bitmask.h>
#include <cpuset.h>

int cpuset_pin(int relcpu);
  Pin the current task to execute only on the CPU relcpu, which is a relative CPU number within
  the current cpuset of that task.  Also automatically pin the memory allowed to be used by the
  current task to the memory on that same node (as determined by the advanced
  cpuset_cpu2node() function)
  (see the cpuset manpages for more details).
```

One simply passes the relative CPU number into the cpuset_pin function. So for this case study, thread 0 of rank 0 passes in relative CPU 0, while thread 1 of rank 0 passes in relative CPU 3, and so forth. The only other thing one needs at the load step is to link in the cpuset library (add –lcpuset to the link line). This simple pinning "trick" was sufficient to make memory accesses as close to the executing thread as possible and the reduction in non-local memory accesses yielded an approximately 2x speed-up.

For Fortran codes that need to access the cpuset_pin function, here is a C-wrapper [8] for the Fortran interface:

```
#include <bitmask.h>
#include <cpuset.h>
#include <stdio.h>

int
cpuset_pin_(int *p_relcpu)
{
  int rtn = cpuset_pin(*p_relcpu);
  if (rtn  < 0) {
    perror("cpuset_pin");
    fprintf(stderr, "cpuset_pin failed for relative cpu %d\n", *p_relcpu);
  }
  return rtn;
}
```

This case study is one of many scenarios where explicit process-pinning yields performance gains. Further optimization steps for the fvGCM code increased the number of threads for each process to use the otherwise idle CPUs in the cpuset.

5 Case Study 3: OpenMP scaling

An often-heard "complaint" from our users is that their code is not scaling as well on the Altix as it was on the Origins. If the code scaled well to hundreds of threads, it was probably run on an SGI Origin. We've had SGI Origins at NASA Ames for over 7 years. Compared to the last SGI Origin in our series, which had a 600 MHz clock and a peak of 1.2 GFlops/processor, the SGI Altix with a 1.5 GHz clock is 5 times faster when comparing peak processor speed. However, the Numalink interconnect has not improved that much. So while the serial version of the code may actually run 5 times faster or more than the Origin, the parallel version may only run 3 times faster or less as one scales to more and more CPUs compared to the Origins. But it's still running faster on the Altix, right? And it's precisely because it is running faster that it doesn't scale as well. To improve scaling on the Altix, one needs to further reduce memory contention and increase locality of memory access. The keyword here is "further." The code may already be well-tuned for a large SMP, but we'll show a "trick" here that will give a bigger performance boost on the Altix than, say, on an Origin.

Here's version 1 of the code:

```
      program main_v1
      parameter(nmax=1000, kmax= 512)
      real (kind=8) :: a,b
      common /block/ a(nmax,nmax), b(nmax,nmax,kmax)
      real (kind=8) :: psum(kmax)
      call random_number(a)        ! fill a with random numbers

!$OMP PARALLEL DO SHARED(b)
      do k = 1,kmax
        b(:,:,k) = 0.0
      enddo
!$OMP END PARALLEL DO
      niter = 40
      do iter = 1,niter
```

```
!$OMP PARALLEL SHARED(a,b,psum,iter)
!$OMP DO
      do k = 1,kmax
        call fillb(nmax,a,b,k,iter)        ! memory contention on a
      enddo
!$OMP END DO
!$OMP DO
      do k = 1,kmax
        call work(nmax,kmax,b,k,psum(k))
      enddo
!$OMP END DO
!$OMP END PARALLEL
      enddo
! dummy print statement to avoid compiler optimizing away code
      if (a(1,1) .lt. -0.1) print *, psum
      end

      subroutine fillb(nmax,a,b,k,iter)
      real (kind=8) :: a(nmax,nmax), b(nmax,nmax,*)
      do j = 1,nmax
        do i = 1,nmax
          b(i,j,k) = (a(i,j) + iter) * k
        enddo
      enddo
      return
      end

      subroutine work(nmax,kmax,b,k,psum)
      real (kind=8) :: b(nmax,nmax,kmax), psum
      psum = 0.0
      do j = 2,nmax-1
        do i = 2,nmax-1
          psum = psum +
     &       0.5 * (b(i+1,j+1,k) + b(i-1,j+1,k) - 2.*b(i,j+1,k)
     &            +  b(i+1,j,k)    + b(i-1,j,k)     - 2.*b(i,j,k)
     &            +  b(i+1,j-1,k) + b(i-1,j-1,k)  - 2.*b(i,j-1,k))
        enddo
      enddo
      return
      end
```

The code has two large arrays “a” and “b”. “a” is 1000 by 1000, and “b” is even larger at 1000 by 1000 by 512 – all real*8’s. The idea here is that “a” is a global array that is used throughout the code to fill array “b”. For real codes, array “a” could be the global variables in the program. In physics, it could be Planck’s constant, the speed of light, mass of the electron, and so forth. In chemistry, it may be the mass of the hydrogen atom, carbon atom, Avogadro’s number, or maybe it could be a look-up table that’s used for interpolating points in some CFD application. In this example, we just fill “a” with random numbers. Now, “a” resides in the memory of the master thread on relative node 1. When the other threads need to use “a”, they need to come to the memory on the first node to get another copy of “a” because it doesn’t fit in cache. “a” is 1000 by 1000 real*8’s or about 8 Mbytes, whereas the Altix 3700 L3 cache is only 6 Mbytes in size. Array “b” is properly initialized in parallel with each thread

initializing an i-j plane that it then uses. The "iter" loop is the main loop of the program, which is iterated 40 times. The arrays "a", "b", and "psum" are all shared arrays. The "iter" do loop control variable is also shared and is passed into the fillb routine so that "b" is different for each iter iteration. After "b" is filled in parallel, it is then used to do some work, also in parallel.

The code then ends with a print statement to prevent the compiler from optimizing code away. In subroutine "fillb", the array "a" and scalars "iter" and "k" are used in forming an i-j plane of "b". Note again that each thread needs to get a fresh copy of array "a" because it does not fit in cache – and that this causes memory contention. Subroutine "work" takes various elements of "b" for particular i-j planes and computes "psum".

Figure 1 shows the scaling chart for version 1 of the program. The code does speed-up with up to 4 threads, but beyond 4 threads, the performance gets progressively worse.

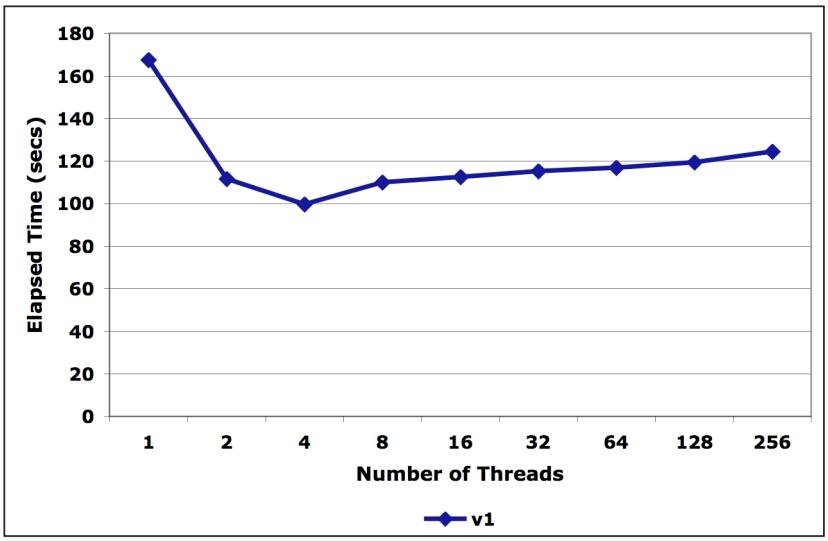

Figure 1: OpenMP scaling for version 1 of the code.

To avoid the memory contention, there is really only one correct way to do that, and that is to make a private copy of array "a" for each thread. Furthermore, the private copy needs to persist from one parallel region to the next, so it needs to be put into a common block, which is made threadprivate. Here are the modified sections of the code:

```
      common /block2/ acopy(nmax,nmax)
!$OMP THREADPRIVATE (/block2/)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!$OMP PARALLEL SHARED(a,b)
            acopy = a    ! make a private copy of a for each thread
!$OMP DO
            do k = 1,kmax
              b(:,:,k) = 0.0
            enddo
!$OMP END DO
!$OMP END PARALLEL
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      call fillb(nmax,acopy,b,k,iter)  ! pass acopy not a
```

So, in the very first parallel region where "b" is initialized, the private copies of array "a" are also made. Of course, there is memory contention here, but it happens only once. And it is "acopy", not "a",

that is passed to "fillb" in the main loop. There are actually many wrong ways to make private copies of array "a". Instead of putting it in a threadprivate common block, one could create private copies in the main parallel loop:

```
      niter = 40
      do iter = 1,niter
!$OMP PARALLEL SHARED(a,b,psum,iter) PRIVATE(acopy)
      acopy = a       ! expensive operation repeated niter times
!$OMP DO
      ...
```

But this is an expensive operation that is repeated niter or 40 times by each thread. Instead of creating another array named "acopy", one might consider making "a" firstprivate. Well, that's almost as bad as this because to make private copies of "a", each thread except for the master thread needs to go to that first node to get a copy of "a" and store it into its local memory.

Figure 2 shows a comparison of OpenMP scaling for versions 1 and 2 of the code. With version 2, the OpenMP scaling is much better. There's more than a factor of 12x improvement at 256 threads. In fact, if instead of iterating 40 times in that main loop, we made niter equal to 1000 to amortize the serial time in calling the random number generator, then version 2 of the code would scale beyond 256 threads and the improvement factor over version 1 would be greater than 12x.

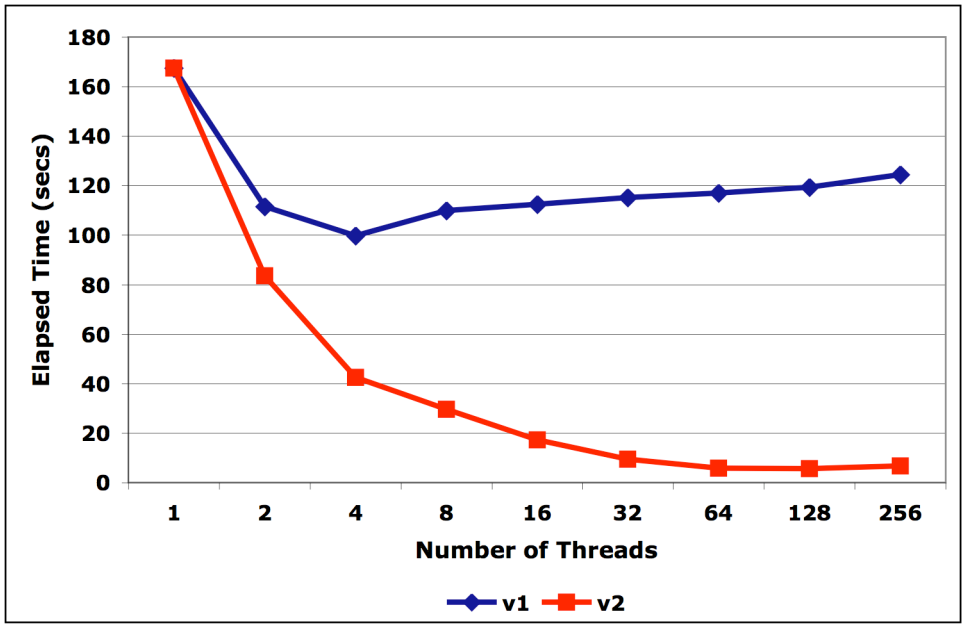

Figure 2: OpenMP scaling comparison for versions 1 and 2 of the code.

Lastly, in Figure 3, we show the speed-up for versions 1 and 2 of the code on the Altix vs. the Origin. For version 1 of the code, the SGI Altix runs about 7 times faster than the Origin for the serial runs, but then drops to a disappointing 27% speed-up at 256 CPUs. For version 2 of the code, the performance gain on the Altix is much better. At 256 CPUs, the speed-up on the Altix is an impressive factor of 11x. From 4 threads to 8 threads, there is a decrease in speed-up factor. This is because the creation of the private copies of "a" – that is, "acopy" – must now go through NUMAlink 3 which has half the bandwidth of NUMAlink 4. Recall that 4 CPUs on a C-brick can communicate with each other via NUMAlink 4, whereas communication to processors outside the C-brick goes through NUMAlink 3. Overall, the speed-up of the runs on the Altix over that of the Origin is more than a factor of 6x through

the whole range of CPUs investigated. This highlights the point that additional tuning to remove memory contention provides a bigger performance boost on the Altix than on the Origin.

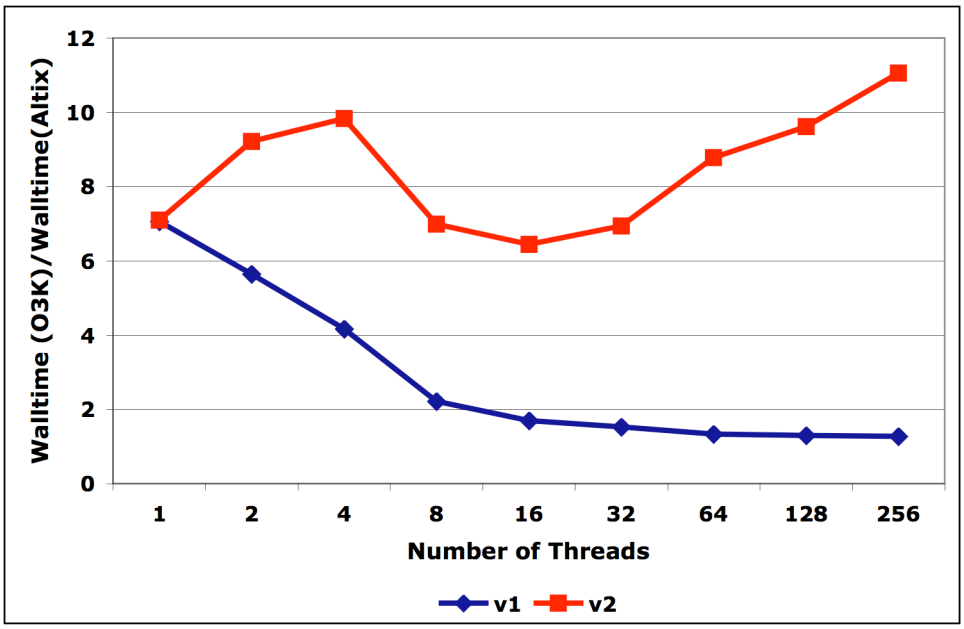

Figure 3: Speed-up on the Altix over the Origin 3000 for versions 1 and 2 of the code.

6 Case Study 4: Unaligned memory access

For the last case study, we'll look at the issue of unaligned memory access. A few months ago, we discovered that two jobs with unaligned access problems could actually interfere with each other and make both jobs slow down even though they are running on different cpusets on the same host. We'll look at the origin of this interference problem and how to detect it from the system's point of view. More importantly, we'll explain how a user can detect and fix unaligned access problems in their code. We'll also show a fix to the kernel developed by SGI, which reduces or eliminates the interference problem. But, first, we'll show a code [9] that demonstrates the unaligned access problem.

```
program prog3
integer, parameter::len_i = 2**20, len_y = len_i/2
common/data/i1,r2(len_i)
real(kind=8)y(len_y)
integer(kind=4) time1, time2, time3

write(6,'(a,1x,z16)')'loc(r2) = ',loc(r2)
call random_number(y)      ! initialize arrays
r2 = 0

call system_clock(time1)
do i = 1,500
call sub(y,len_y)          ! properly aligned on 8-byte boundaries
enddo
call system_clock(time2)
do i = 1,500
call sub(r2,len_y)         ! unaligned memory access
```

```
enddo
call system_clock(time3)
write(6,"('times = ',g12.6,1x,g12.6)")time2-time1,time3-time2
end

subroutine sub(x,len)
real(kind=8)x(len)
do i = 1,len
x(i) = i * x(i)        ! a load and store into same memory location
enddo
end
```

There are two arrays in this code: r2 and y. "r2" is of length 2**20 or about a million, and "y" is half that length. A common block is used to purposely ensure that r2, an array of real*4's, is aligned on a 4-byte boundary because the integer "i1" is aligned on an 8-byte boundary. The expression "to be aligned on a 4-byte boundary" means that the memory address is divisible by 4 but not by 8. "y" is properly aligned on an 8-byte boundary, so when it is passed into subroutine sub, all the loads and stores are aligned. However, when r2 is passed into sub, all the loads and stores are unaligned, that is they are aligned on 4-byte boundaries but not 8-byte boundaries. Each unaligned access causes a kernel interrupt to form an 8-byte number out of 2 neighboring 4-byte quantities. This code prints out the memory location of the beginning address of r2 to verify that it's indeed aligned on a 4-byte boundary and the times (time2 – time1) for aligned access versus times (time3 – time 2) for unaligned access.

There are a couple of other things to point out about this code. First, note that it is "len_y" that is passed into subroutine sub for both arrays y and r2. This is to enable a direct timing comparison of the same number of loads and stores for both aligned and unaligned access. Secondly, r2 is a real*4 array and subroutine sub is expecting a real*8 array. For the vast majority of codes, this would be a programming bug. However, this is legal Fortran, and one can consider the declaration of r2 in the common block as simply a storage unit. Interestingly enough, this precise scenario was used in SGI's MPT library for the MPI_Recv function [8]. In the C version of the MPI_Recv function, there is a "status" pointer to a structure of type MPI_Status. Because the original MPI standard was written to the Fortran77 specification [10] (not Fortran90), there was no standard conforming way to define a similar structure in Fortran. As a result, the MPI_status type is defined in Fortran to be an array of integers of a certain length. In the SGI implementation, one of the fields of the MPI_Status type is an 8-byte integer (to accommodate the needs of larger memory machines), and was formed from two consecutive 4-byte integers. The Fortran array of integers, however, only guarantees 4-byte alignment and not 8-byte alignment. This turned out to be the cause for the vast majority of the unaligned access problems experienced by MPI codes running on our Altix. After this fact was discovered, SGI has provided a fix to the MPT library, which is currently being used as the default MPT module on the Columbia supercomputer. The fix was done by changing the Fortran interface routines to memcpy the incoming array of ints into a properly aligned MPI_Status variable on entry, and then copy it back out again upon return [8]. But, this is getting ahead of the story.

When the program "prog3" is run on SGI's ProPack 4.2, which uses a Linux 2.6 kernel, one sees an output similar to the following:

```
loc(r2) =  6000000000418CD4
times =        3976      3289999
```

The loc of r2 is written out in hexadecimal notation. From the last digit, one can see that r2 is aligned on a 4-byte boundary. Also, the times for unaligned access are about 800 times longer than for aligned access. Furthermore, if this code is run interactively, the following messages would be scrolling on the screen:

```
prog3(13657): unaligned access to 0x6000000000418cd4, ip=0x4000000000002ff0
prog3(13657): unaligned access to 0x6000000000418cdc, ip=0x4000000000002ff0
prog3(13657): unaligned access to 0x6000000000418ce4, ip=0x4000000000002ff0
prog3(13657): unaligned access to 0x6000000000418cec, ip=0x4000000000002ff0

(and 5 seconds later…)

prog3(13657): unaligned access to 0x6000000000726974, ip=0x4000000000002ff0
prog3(13657): unaligned access to 0x600000000072692c, ip=0x4000000000003000
prog3(13657): unaligned access to 0x600000000072697c, ip=0x4000000000002ff0
prog3(13657): unaligned access to 0x6000000000726934, ip=0x4000000000003000

(and so on …)
```

The message contains the executable name, the pid, the location of the unaligned access, and the instruction pointer. One can see that the first address is the beginning location of r2 and the subsequent addresses are spread 8-bytes apart. The writing of these unaligned access messages is throttled to a maximum of 4 messages every 5 seconds. If the code is not run interactively, then there is no tty connected to the job, and these messages would be logged in the /var/log/messages file.

We look at the /var/log/messages file quite often in trouble-shooting user problems. We had seen lots of these unaligned access messages before and thought that they were mostly a nuisance in making it more difficult to find the more important messages logged by the kernel, until a user started complaining that her job took twice as long to run after the operating system was changed from the Linux 2.4 kernel to the Linux 2.6 kernel. The 2.4 kernel uses the RedHat Enterprise Linux Advanced Server 3 operating system, which does not log unaligned messages and the 2.6 kernel uses SuSE Linux Enterprise Server 9 (SLES9), which does log messages. At the time that user was running her job, which was running at half the expected speed, there was only one other job from another user running on the system. Both jobs were logging an inordinate amount of unaligned access messages in the /var/log/messages file. We didn't think that two jobs with unaligned access problems could interfere with each other until we ran the following experiment.

1, 2, 4, 8, and 16 concurrent copies of the "prog3" program were run on a Columbia 512-processor host that had the ProPack 4.2/Linux 2.6 kernel. Figure 4 shows the elapsed time for running "prog3" when multiple copies of "prog3" are run at the same time. With just one copy, it takes about 5 minutes, with 2 copies, about 10 minutes, with 4, about 20 minutes, and so on. There's clearly interference when running multiple copies. This doesn't have to be multiple jobs running concurrently, it could even be a single MPI job where the various processes are interfering with each other. All of this increase in elapsed time is due to increases in system time.

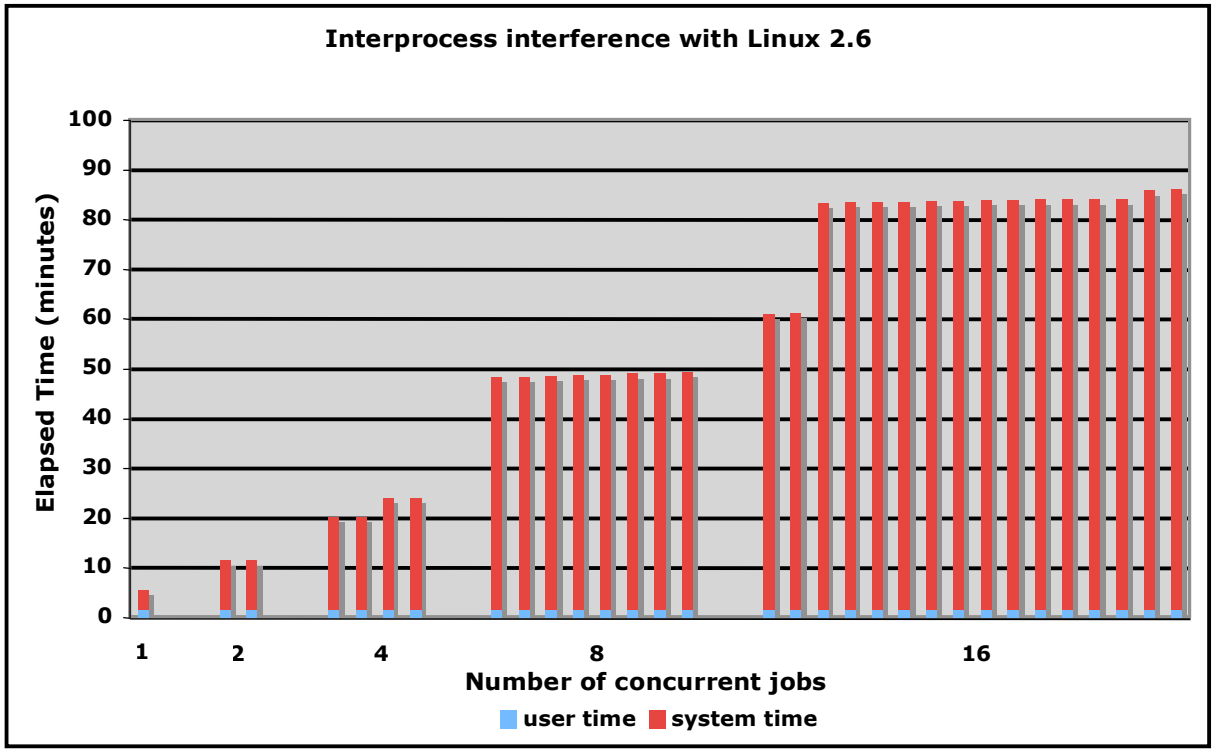

Figure 4: Unaligned memory accesses cause interprocess interference with Linux 2.6.

Figure 5 shows the results of the same experiment obtained from a Columbia 512p host running ProPack 3.6 and the Linux 2.4 kernel. There is absolutely no inter-process interference with the older operating system, and all the runs completed in under 3 minutes, which is less time than a single run on ProPack 4.2. These experiments were key to convincing SGI engineers that there was an unaligned access interference problem.

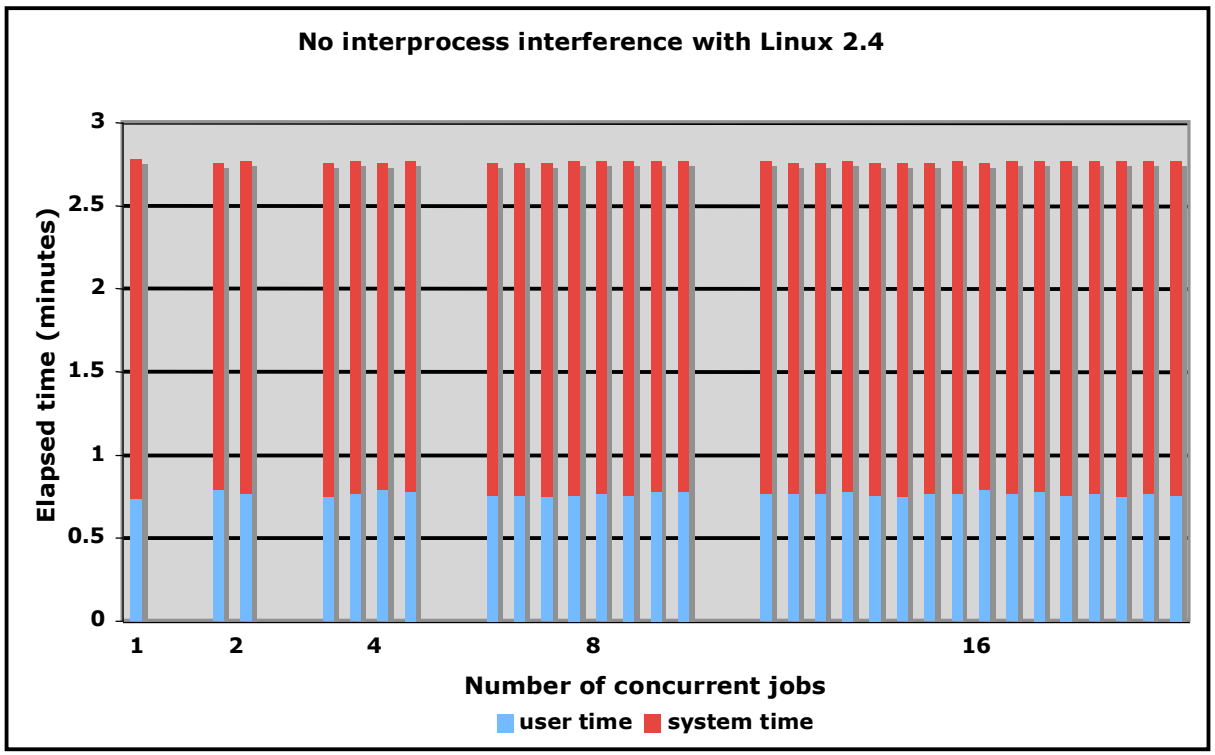

Figure 5: Unaligned memory accesses do not cause interprocess interference with Linux 2.4.

Right after that user complained about her code running slowly, one of our local SGI engineers [11] profiled the system. Here's a two line script that he ran as root to profile system activity on CPUs 10 to 30:

```
cp /boot/System.map-`uname -r` ./System.map
cpuset -i /PBSPro -l profile.pl -- --no_dplace  -c10-30 /bin/sleep 300
```

A copy of the System.map file is necessary in the local directory to profile the kernel. The cpuset command in the script creates a cpuset consisting of CPUs 10-30. This cpuset is overlaid on top of CPUs already pre-assigned to the other user job by the PBS batch scheduler (and the creation of an overlaying cpuset on top of another user's cpuset is the primary reason why this script must be run as root).

Profile.pl is a Perl script that eventually uses pfmon to get profiling information. Here's the output from running the profiling script:

```
Profiling output:
user ticks:          331447        57.21 %
kernel ticks:        247947        42.79 %
idle ticks:          3             0 %

Using ./System.map as the kernel map file.
=========================================
                    Kernel

     Ticks     Percent  Cumulative   Routine
                         Percent
-----------------------------------------------------------------
    244901       98.77    98.77     within_logging_rate_limit
       634        0.26    99.03     printk
       621        0.25    99.28     rcu_process_callbacks
                                    …
```

One sees that 43% of the time is spent in the kernel, and of these 43%, approximately 99% of the time is spent inside a routine called within_logging_rate_limit. This function determines whether to log a message or not. The actual logging of the message takes about a quarter of a percent and processing the unaligned access fault takes another quarter of a percent of the kernel time. Everything else is miniscure.

To see why so much time is spent in the within_logging_rate_limit function, we look at the segment of code taken from /usr/src/linux/arch/ia64/kernel/unaligned.c:

```
/*
 * Make sure we log the unaligned access, so that user/sysadmin can notice it and
 * eventually fix the program.  However, we don't want to do that for every access so
 * we pace it with jiffies.  This isn't really MP-safe, but it doesn't really have to be
 * either...
 */
static int
within_logging_rate_limit (void)
{
      static unsigned long count, last_time;  ← count & last_time on hot cache line
```

```
        if (jiffies - last_time > 5*HZ)
                count = 0;
        if (++count < 5) {                          ← count updated every single time!
                last_time = jiffies;
                return 1;
        }
        return 0;
}
```

The problem is that both count and last_time are static variables. "jiffies" is a kernel timing variable measured in units of Hz. When the number of jiffies has incremented past last_time by more than 5 Hz, count is reset to 0. Here, count is incremented for every unaligned access, and as long as count is less than 5, it updates "last_time" and returns 1 to print the message. Now, since both "count" and "last_time" are both static, whenever a process needs to update "count" or "last_time," it needs to invalidate all other processes' copies of that cache line. In the words of kernel hackers, this hot cache line is zipping around the system between processes that have unaligned accesses. And because "count" is updated every single time, the invalidation and contention on the hot cache line has to occur whether an unaligned access message is logged or not.

After we pointed out the problem that unaligned memory accesses can cause interference between concurrently running jobs to SGI engineers, they came up with the following fix, which has now been incorporated into SLES10:

```
static int
within_logging_rate_limit (void)
 {
        static unsigned long count, last_time;

        if (jiffies - last_time > 5*HZ)
                count = 0;
        if (count < 5) {
                last_time = jiffies;
                count++;                ← count updated ONLY if less than 5
                return 1;
        }
        return 0;
}
```

In this new function, count and last_time are updated only if count is less than 5. This fix is enough to eliminate or significantly reduce interference between jobs.

But more important than the kernel fix is to fix the user's code. So how can a user find the source of their unaligned access? There are two methods: The first is that the user can issue the command:

```
prctl --unaligned=signal
```

before running the application. This would cause a core dump at the first instance of an unaligned access. If the code is also compiled with –traceback and –g, then the stack trace will contain both the routine name and line number of the code that is causing the unaligned access. Another method is to compile and link the code with the following flag:

```
-Wl,--print-map
```

This will pass the --print-map option to the loader to print the loadmap. Then, one can track down the addresses given by those unaligned access messages via the loadmap down to the corresponding variables within the code.

7 References

[1] Y.-T. Chang and J. Chang, Getting Good Performance on OpenMP and Hybrid MPI+OpenMP Codes on SGI Altix, SGIUG 2005 Technical Conference and Tutorials, June 13-16, 2005, Munich, Germany.

[2] J. Chang, Columbia Application Performance Tuning Case Studies, SGIUG 2006 Technical Conference and Tutorials, June 5-9, 2006, Las Vegas, Nevada.

[3] SGI Altix 3000, http://www.sgi.com/products/servers/altix/3000/

[4] October 26, 2004 press release, http://www.sgi.com/company_info/newsroom/press_releases/2004/october/worlds_fastest.html, http://news.com.com/SGI+claims+lead+in+supercomputer+race/2100-1010_3-5426813.html?tag=nl

[5] November 5, 2004 press release, http://news.com.com/IBM+set+to+take+supercomputing+crown/2100-1010_3-5439523.html

[6] Top500, http://www.top500.org

[7] B.-W. Shen, R. Atlas, J.-D. Chern, O. Reale, S.-J. Lin, T. Lee, J. Chang, The 0.125 degree finite-volume general circulation model on the NASA Columbia supercomputer: Preliminary simulations of mesoscale vortices, Geophys. Res. Lett., 33, L05801, doi:10.1029/2005GL024594 (2006). http://www.agu.org/pubs/crossref/2006/2005GL024594.shtml

[8] Bron Nelson, private communication.

[9] Art Lazanoff, private communication.

[10] MPI Standard, http://www-unix.mcs.anl.gov/mpi/

[11] Scott Emery, private communication.